

Outline

1. Launching a Spark cluster on AWS
2. HDFS, YARN and Spark
3. First steps with Spark
4. Map-and-reduce architecture

Création d'un cluster Spark sur AWS

- ☐ Une fois connecté à la console AWS cherchez le service EMR pour *Elastic Map Reduce*. C'est la *Plateforme as a Service* qui permet de créer un cluster Hadoop dans AWS. Vous allez juste choisir la configuration de votre cluster, et AWS va créer toutes les VM, les mettre en réseau et installer toutes les applications choisies pour vous. Créer un cluster Hadoop à la main est laborieux et n'est pas réellement intéressant, c'est pourquoi les divers fournisseurs de services cloud proposent de telles PaaS
- ☐ Vous allez arriver sur un écran similaire à celui-ci

Bienvenue dans Amazon Elastic MapReduce

Amazon Elastic MapReduce (Amazon EMR) est un service Web qui permet aux commerces, aux chercheurs, aux analystes de données et aux développeurs de traiter de grandes quantités de données de manière simple et économique.

Apparemment, vous n'avez aucun cluster. Créez-en un maintenant :

[Créer un cluster](#)

Fonctionnement d'Elastic MapReduce

Charger : Téléchargez les données et l'application de traitement sur S3.

Créer : Configurez et créez votre cluster en spécifiant les entrées et les sorties de données, la taille du cluster, les instances.

Contrôle : Surveillez l'état et la progression de votre cluster. Récupérez la sortie des données.

Informations supplémentaires

En savoir plus sur Elastic MapReduce :

- [Présentation d'EMR](#)
- [FAQ](#)
- [Tarification](#)

En savoir plus sur l'utilisation d'Elastic MapReduce :

- [Forum](#)
- [Documentation](#)
- [Manuel du développeur](#)
- [Référence API](#)
- [EMR sur GitHub](#)
- [Portail d'aide](#)

Les fois suivantes il ressemblera à cela

Créer un cluster [Afficher les détails](#) [Cloner](#) [Réinitialiser](#)

Filtre : Tous les clusters 4 clusters (tous chargés)

	Nom	ID	Statut	Heure de création (UTC+1)	Temps écoulé	Heures normales
<input type="checkbox"/>	Mon cluster	j-1H72GTU5MWKU2	Résilié Demande utilisateur	23-03-2021 15:23 (UTC+1)	53 minutes	24
<input type="checkbox"/>	Mon cluster	j-38AR1CNY33PPB	Résilié Demande utilisateur	23-03-2021 15:15 (UTC+1)	11 minutes	24
<input type="checkbox"/>	NotebookCluster	j-39CG54L47FWG	Résilié Demande utilisateur	23-03-2021 14:52 (UTC+1)	13 minutes	8
<input type="checkbox"/>	Mon cluster	j-2P3UZ7CHZ3BHC	Résilié Demande utilisateur	23-03-2021 14:44 (UTC+1)	9 minutes	24

Dans tous les cas cliquez sur **Créer un cluster**

- ☐ Sur la page suivante voici les configurations à saisir :
 - ☐ Nom du cluster : le nom que vous souhaitez. Le nom par défaut convient très bien
 - ☐ Journalisation : décochez ce paramètre. Il permet de sauvegarder les *log* (journaux) de votre cluster pour ensuite aller chercher la source d'une erreur. Mais pour ce TP cela ne sera pas utile

- ☐ Mode de lancement : `cluster`. La différence entre `cluster` et `Exécution d'étape` est que `cluster` permet d'interagir avec le cluster, alors que `Exécution d'étape` crée un cluster, lance un ou plusieurs scripts et s'arrête. C'est parfait quand vous voulez lancer un "job" sur vos données. Le cluster se lance, fait les calculs puis va exporter les résultats et s'éteindre tout seul.
- ☐ Libérée (Release en VO) : `emr-5.31.0`, donc l'avant dernière version 5.XX. La dernière a des problèmes pour l'utilisation de notebook.
- ☐ Type d'instance : des `m5.xlarge` conviennent parfaitement. Si vous voulez vous pouvez essayer des machines plus puissantes, mais cela ne va pas impacter fortement les temps de calculs.
- ☐ Nombre d'instance : `3`, mais vous pouvez essayer avec plus d'instance (limitez vous à 6).

Pour vous donner une idée des prix unitaire des machines voici un tableau des instance m5.XX. Pour le prix du cluster, multipliez par le nombre de machine

Instance	Prix unitaire par heure
m5.xlarge	0.24\$
m5.2xlarge	0.48\$
m5.4xlarge	0.96\$
m5.8xlarge	1.86\$

- ☐ Paire de clé EC2 : sélectionnez la clé du TP 0. Si vous n'en avez pas sélectionnez `sans paire de clé EC2`.
- ☐ Puis cliquez sur `Créer un cluster`

Voici ce à quoi vous devez arriver :

Configuration générale

Nom du cluster

☐ Journalisation ⓘ

Mode de lancement ☒ Cluster ⓘ ☐ Exécution d'étape ⓘ

Configuration des logiciels

Libère

Applications

- ☐ Core Hadoop: Hadoop 2.10.0, Hive 2.3.7, Hue 4.7.1, Mahout 0.12.6, Pig 0.17.0, and Tez 0.9.3
- ☐ HBase: HBase 1.4.13, Hadoop 2.10.0, Hive 2.3.7, Hue 4.7.1, Phoenix 4.14.3, and ZooKeeper 3.4.14
- ☐ Presto: Presto 0.238.3 with Hadoop 2.10.0 HDFS and Hive 2.3.7 Metastore
- ☒ Spark: Spark 2.4.6 on Hadoop 2.10.0 YARN and Zeppelin 0.8.2

☐ Utiliser AWS Glue Data Catalog pour les métadonnées de table ⓘ

Configuration du matériel

Type d'instance Le type d'instance sélectionné ajoute un volume EBS GP2 par défaut de 64 GiO par instance. [En savoir plus](#)

Nombre d'instances (1 nœud maître et 2 nœuds principaux)

Cluster scaling ☐ scale cluster nodes based on workload

Sécurité et accès

Paire de clés EC2 ⓘ [Apprenez à créer une paire de clés EC2](#)

Autorisations ☒ Par défaut ☐ Personnalisé

Utilisez les rôles IAM par défaut. Si des rôles sont absents, ils seront créés automatiquement pour vous avec des stratégies gérées pour les mises à jour automatiques de stratégies.

Rôle EMR [EMR_DefaultRole](#) ⓘ

Profil d'instance EC2 [EMR_EC2_DefaultRole](#) ⓘ

[Annuler](#) [Créer un cluster](#)

La création d'un cluster prend plus de temps que la création d'une machine unique (de l'ordre de quelques minutes). Car AWS doit lancer X machines avec des configurations lourdes, les mettre en réseau etc.

- ☐ Une fois le cluster passer en "En attente" ou "Stand by", allez dans blocs-notes, puis cliquez sur **Créer un bloc-notes**

Amazon EMR

EMR on EC2

Clusters

Blocs-notes

Git repositories

Configurations de la sécurité

Bloquer l'accès public

Sous-réseaux VPC

Événements

EMR on EKS

Virtual clusters

Blocs-notes

Utilisez des blocs-notes Jupyter gérés par EMR pour analyser les données de façon interactive avec du code en direct, des textes narratifs, des visualisations, et bien plus. Les blocs-notes sont exécutés gratuitement et sont enregistrés dans Amazon S3 indépendamment des clusters. La facturation standard pour les clusters et Amazon S3 s'applique.

[Créer un bloc-notes](#) [Afficher les détails](#) [Ouvrir dans JupyterLab](#) [Ouvrir dans Jupyter](#) [Démarrage de](#) [Arrêter](#) [Supprimer](#)

Filtre : 1 bloc-notes (tout chargé)

Nom	Statut	Clus
test	Arrêté	i-1H

- ☐ Sur l'écran suivant vous allez devoir spécifier :
- ☐ Un nom pour votre *notebook*
 - ☐ Le cluster que vous souhaitez utiliser. Noter que vous pouvez créer à la volée un cluster si vous le souhaitez
 - ☐ Le rôle de sécurité et les groupes associés au service. Nous allons garder les paramètres par défaut
 - ☐ L'endroit où sera stocké votre notebook sur S3. Cela peut servir à recharger un *notebook* fait plus tôt.
 - ☐ Un dépôt git si vous souhaitez versionner votre code.

Voilà quoi devrait ressembler votre écran avant validation.

Créer un bloc-notes

Nommer et configurer votre bloc-notes

Nommez votre bloc-notes Jupyter géré par EMR, choisissez un cluster ou en créez-en un et personnalisez les options de configuration si vous le souhaitez. [En savoir plus](#)

Nom du bloc-notes.* 1
Les noms peuvent contenir uniquement des lettres (a-z), des chiffres (0-9), des traits d'union (-) ou des traits de soulignement (_).

Description 2
256 caractères max.

Cluster* ☒ Choisir un cluster existant 3
 [Mon cluster j-30YZFPZ86VI04](#)

☐ Créer un cluster ⓘ

Groupes de sécurité ☒ Utiliser des groupes de sécurité par défaut ⓘ
☐ Choisir des groupes de sécurité (vpc-e2567a98)

Rôle de service AWS* ⓘ

Emplacement du bloc-notes* Choose an S3 location where files for this notebook are saved.
☒ Use the default S3 location
s3://aws-emr-resources-523967347856-us-east-1/notebooks/
☐ Choose an existing S3 location in us-east-1

► **Référentiel Git** Lien vers un référentiel Git pour enregistrer votre bloc-notes dans un environnement de gestion des versions

► **Balises** ⓘ

* Obligatoire

- ☐ La création du *notebook* doit être rapide. Une fois votre *notebook* prêt, cliquez sur **Ouvrir** dans **JupyterLab**. Cela ouvrira une interface JupyterLab pour rédiger des *notebooks*. Par défaut vous pouvez faire des *notebooks* :

- ☐ Python3
- ☐ PySpark : Spark avec python
- ☐ Spark : Spark en Scala
- ☐ SparkR : Spark en R.

Créez un *notebook* pyspark et exécutez dans un cellule la ligne suivante :

```
1 | spark
```

cela devrait produire la résultat suivant.

```
Untitled.ipynb
[1]: spark


Starting Spark application
ID          YARN Application ID    Kind  State  Spark UI  Driver log  Current session?
0  application_1616589341417_0001  pyspark  idle  Link      Link        ✓

SparkSession available as 'spark'.
<pyspark.sql.session.SparkSession object at 0x7fde917e1b90>

[ ]:
```

Pour des questions de sécurité, les liens vers l'interface Spark (*Spark UI*) et les journaux du driver (*Driver log*) ne fonctionnent pas. Il faut pour y accéder réaliser une connexion SSH + un transfert de port.

Exercice 3. First steps with Spark

 **3.1 Your first DataFrame** — Spark's main object class is the `DataFrame`, which is a distributed table. It is analogous to R's or Python's data frames: one row represents an observation, one column represents a variable. But contrary to R or Python, Spark's `DataFrames` can be distributed over hundred of nodes.

- Run the following:

```
1 df = spark.read.format("parquet")
2   .option("mode", "failFast")
3   .option("header", "true")
4   .option("inferSchema", True)
5   .option("path", "file://path/to/file.csv.gz")
6   .load()
7
8 df.cache()
```

You have just created a data frame! 🎉

Data frames are **immutable**: there is no method to alter one specific value once one is created. Also, data frames are **distributed**: they are split into blocks, ill-named **partitions**¹, that are stored separately in the memory of the workers nodes.

The input file is a parquet file. Parquet is an open source column-oriented format that provide storage optimization. Spark natively can create a `DataFrames` from a parquet file.

Why do we cache the `DataFrame` ? And are there any other solutions ?

- In the Spark console, click on "Show incomplete applications" to see our current `pyspark` session. Look at the timeline. How many **executors** were used to perform the importation?
- Click on the **job** corresponding to the importation, then on the unique **stage** that composes this job. On which nodes of the cluster were the executors located?
- For the importation stage, we have the equivalence one task = one partition, since the task is actually to create one partition. Does the number of tasks executed on each node recall you anything? (*Hint: go back to the HDFS Console.*)
- Lastly, open the event time line. From what you see, how many processors on each nodes were there? Can you confirm it from EC2?

`r emo::ji("wrench")` **3.3 DataFrame manipulation** — Data frames are immutable, but they can be **transformed** in other data frames. Such **transformations** include: filtering, sampling, dropping columns, selecting columns, adding new columns...

- First, you can get information about the columns with:

```
1 flights.columns      # get the column names
2 flights.schema      # get the column names and their respective
  type
3 flights.printSchema() # same, but human-readable
```

- What does the following code do?

```
1 passengers_per_month = flights\
2   .select("PASSENGERS", "YEAR", "MONTH")\
3   .groupBy("YEAR", "MONTH")\
4   .sum("PASSENGERS")
```

- And this one?

```
1 flights_from_2018 = flights\
2   .sample(fraction=0.001)\
3   .filter(flights.YEAR==2018)\
4   .limit(100)
```

- And this one?

```
1 overconfident_carriers = flights\
2   .select("CARRIER", "DEPARTURES_SCHEDULED",
3   "DEPARTURES_PERFORMED")\
4   .withColumn(      # computes new variable
5     "OVERCONFIDENCE", # called "OVERCONFIDENCE"
6     (flights.DEPARTURES_SCHEDULED - flights.DEPARTURES_PERFORMED)/
7     flights.DEPARTURES_PERFORMED
8   )\
9   .groupBy("CARRIER")\
10  .sum("OVERCONFIDENCE")\
11  .sort("sum(OVERCONFIDENCE)")
```

- Run each of the code sections.

`r emo::ji("sleeping")` **3.3 Lazy evaluation**

- What happens when you run `flights`, like you would do in Python or R? Why?
- At question **3.2**, did you get any result at all? Did any of the instructions cause computation to actually happen? (*Hint: look at the Spark console*)

This is because Spark has what is known as **lazy evaluation**, in the sense that it will wait as much as it can before performing the actual computation. Said otherwise, when you run an instruction such as:

```
1 filtered_flights = flights.filter(flights.YEAR==2018)
```

... you are not executing anything! Rather, you are building an **execution plan**, to be realised later.

Spark is quite extreme in its laziness, since only a handful of methods called **actions**, by opposition to **transformations**, will trigger an execution. The most notable are:

1. `collect()`, explicitly asking Spark to fetch the resulting rows instead of to lazily wait for more instructions,
2. `take(n)`, asking for `n` first rows
3. `first()`, an alias for `take(1)`
4. `show()` and `show(n)`, human-friendly alternatives ²
5. `count()`, asking for the numbers of rows
6. all the "write" methods (write on file, write to database)

This has advantages: on huge data, you don't want to accidentally perform a computation that is not needed. Also, Spark can optimize each **stage** of the execution in regard to what comes next. For instance, filters will be executed as early as possible, since it diminishes the number of rows on which to perform later operations. On the contrary, joins are very computation-intense and will be executed as late as possible. The resulting **execution plan** consists in a **directed acyclic graph** (DAG) that contains the tree of all required actions for a specific computation, ordered in the most effective fashion.

This has also drawbacks. Since the computation is optimized for the end result, the intermediate stages are discarded by default. For instance, in the following:

```
1 # step 1
2 flights_overconfidence = flights\
3   .withColumn(
4     "OVERCONFIDENCE",
5     (flights.DEPARTURES_SCHEDULED - flights.DEPARTURES_PERFORMED)/
6     flights.DEPARTURES_PERFORMED
7   )
8 # step 2
9 flights_overconfidence_2018 = flights_overconfidence\
10  .filter(flights.YEAR==2018)\
11  .collect()
```

... the intermediate `flights_overconfidence` does not exist more after `collect()` have been called than before the call. Indeed, the values for other years than 2018 have not been computed at all!

- Now run:

```
1 passengers_per_month.show()
2 flights_from_2018.count()
3 overconfident_carriers.take(10)
```

Was something executed this time?

- You can get the execution plan from the Spark console, or from Python with the `explain()` method. Try with `flights_from_2018.explain()`. Does the order of the stages make sense?

`r emo::ji("billed_hat")` **3.4 Practice**

The complete list of methods (transformations and actions) for data frames is listed [here](#). The aggregation functions, such as `sum()`, `max()`, `mean()` ... are listed [here](#).

- What are the 10 biggest airports of the USA in 2018?
- What is the longest regular flight served by each carrier in 2000?

Exercise 4. Map-and-reduce architecture

Manny computation algorithms can be expressed using two stages:

- a **map stage**, where the instructions can be applied element-wise, in the sense that if elements are arranged as a list, the operation on element e , does not depend on the value of e'

- a **reduce stage**, where the instructions obtained in the first stage are combined pairwise recursively ; each time a result is obtained from the first stage, it is combined with earlier results, as in an accumulator

The reduce function must be associative, and commutativity simplifies the reduce step even further. Typical examples are addition and multiplication. Concatenation is associative, but not commutative.

`r emo::ji("man_teacher")` 4.1. Map-and-reduce examples

- Find two examples of computation problems that decompose well under the map-and-reduce principle, and one that can't.
- The `count()` method is expressible as a *map-and-reduce* algorithm. `flights.count()` is equivalent to the following code. Can you make clear how the job is executed? Is it faster?

```
1 # the map function is not available at the data frame level
2 # we have to go down at the data set (RDD) level
3 flights\
4   .rdd\
5   .map(lambda flight: 1)\
6   .reduce(
7     lambda accumulator, value:
8       accumulator + value
9   )
10 # reduce is an action verb
11 # we do not need an explicit collect()
```

- Explain the `lambda flight: 1` syntax. How do you call this kind of object?
- Look at the Spark console to see where the different stages of the computation actually happened.

`r emo::ji("soccer")` 4.2. Practice

- Compute the total number of passengers transported following the map-and-reduce principle. Is it faster than
- What does the following code do?

```
1 def my_function( a, b ) :
2   return b if b > a else a
3
4 flights\
5   .rdd\
6   .map(lambda flight: flight.AIR_TIME)\
7   .reduce( my_function )
```

- The *map* stage may as well return a tuple (FR: n-uplet), as long as the you have an corresponding well chosen *reduce* stage. For instance, what does the following do?

```
1 flights\
2   .rdd\
3   .map(lambda flight: (flight.AIR_TIME, flight.CARRIER))\
4   .reduce(lambda a, b: a if a[0] > b[0] else b)
```


- How would you recode the `mean()` function in two successive map-and-reduce operations? Is it possible with only one?
 - What about the variance?
-

1. Usually a "partition" is an set of chunks that cover all the data, without any repetition between the chunks. But not in Spark! [↗](#)
2. `first()` is exactly `take(1)` ([ref](#)) and `show` prints the result instead of returning it as a list of rows ([ref](#)) [↗](#)